CS 2413 Data Structures – Programming Project 4
Due April 15th, 2018, 11:59 PM

Objectives

1. Use C++ STL "`string`" class in your program.
2. Use C++ STL "`list`" class in your program.
3. Implement "`ttree`" and "`tnode`" classes.
   (a) insert() method of "`ttree`".
   (b) search() method of "`ttree`".
   (c) ostream operator of "`ttree`".
   (d) Other helper methods for you to implement these classes (default constructor, destructor, accessors/getters, mutators/setters, etc)
4. Runtime: Your programs correctly read the input, construct the tree and generate the output as specified.
5. Document your project thoroughly as the examples in the textbook. This includes but not limited to header comments for all classes/methods, explanatory comments for each section of code, meaningful variable and method names, and consistent indentation.

Project Description

When you are texting as you type character by character, the words that match the characters you typed are displayed. In this project, you will implement the data structure "`ttree`" for this functionality. You will also learn how to use the string class and list (linked list) class in C++ Standard Template Library (STL).

Exception Handling

You will implement the following exception handling classes in ttree.

```
class ttreeException: public exception{};
class ttreeNotFound: public ttreeException{};
```

C++ Standard Template Library

The C++ STL contains a set of templated container classes, algorithms, functions and iterators that you can use in your programs. Search online to learn about <strings> and <list> classes, corresponding methods, iterators and use them in this project. You will also use the methods for substring matching from the STL string class.

How the "ttree" Works?

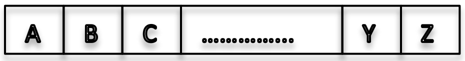
When a "ttree" is created, a maximum depth is given. This value limits the number of levels of the tree, which has a **default value of 5** in this project. Each ttree node contains an array of tnodes (`_tnodes`) with the size of 26, representing A(0) to Z(25). It also has a data field to keep tracking the depth of current node. There are two sets of data will be stored in a tnode: _nextLevel (of ttrees) and _words. We will explain these later in the examples.

To convert a character's ASCII value so that it can be used to index into an array we have provided the code below:

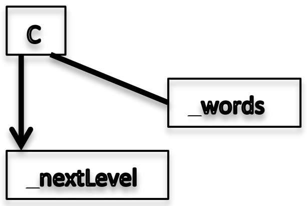
```
#include <iostream>
using namespace std;

int main ()
{
  int i;
  char c = 'Z';
  i = ((int) c) - 65 // Or use (c - 'A');
  cout << "The ascii value of " << c << " is " << i << endl;
}
```

For the convenience of explanation, we will simply use the following diagram to represent a "level" of the ttree where each of the "A", "B", "C" …… "Z" is a tnode. **Note that when we use letters A, B, etc in the arrays we actually are indexing into the array positions that the letter A corresponds to.**

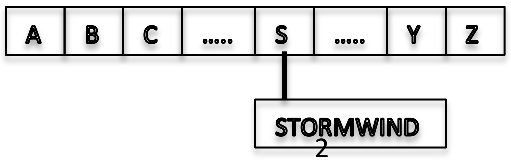


And the following diagram shows a "tnode" that will be used in the explanation. The line with an arrow points to the next level of this tnode (character C in this example) and the line without the arrow point to the words stored in this tnode.



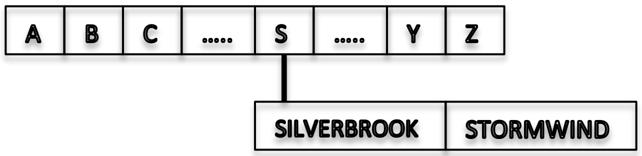
Initially, the tree is empty and hence `_nextLevel` and `_word` are set to `NULL`. If these values are NULL, we will not draw anything in the diagram for simplicity. We assume the maximum level of the tree is 3 in the example. Say now we want to insert the following words into the tree in the order: STORMWIND, SILVERBROOK, BLACKROCK, BLACKWIND, BLADESPIRE and STONARD (we assume the words are all in capital (uppercase) letters in this project). The resulting tree after each insertion is shown below.

STORMWIND



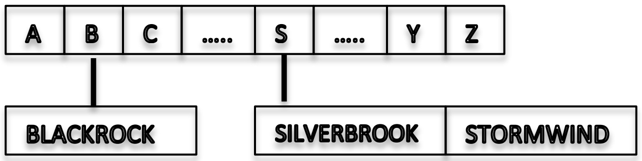
Since STORMWIND starts with 'S", we look into the 19th element (`_tnode[18]`) in the array. Because there's nothing stored in it, so its `_nextLevel` and `_words` are both NULL. This means we can simply initialize `_words` and store "STORMWIND" in this node.

SILVERBROOK



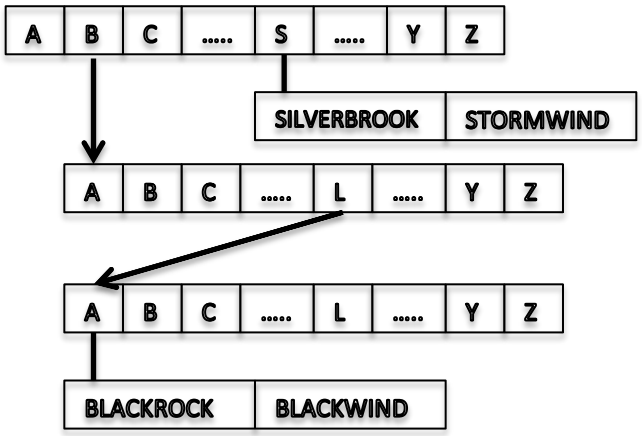
Similarily, SILVERBROOK should go to the same tnode as STORMWIND because they both start with the letter 'S'. Now we check both words and *the length of common prefix* is 1. This means they are staying at the same level. Since `SI < ST`, we put SILVERBROOK in front of STORMWIND.

BLACKROCK



Similarily, we insert BLACKROCK into its corresponding tnode (index = 1).

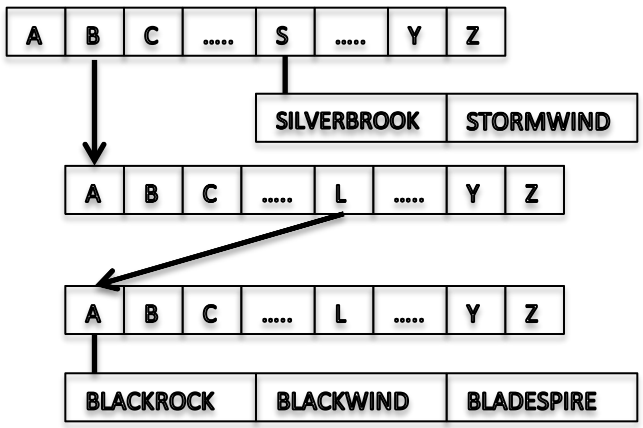
BLACKWIND



Next is to insert BLACKWIND and it should go to `_tnode[1]`. Now we compare BLACKWIND with BLACKROCK and there are five prefix characters in common. Hence we will need to create 4 new levels.

However, we will only create two new levels because *we assume that the maximum level in this example is 3*. Then we add both to `_tnode[0]` at the 3rd level.

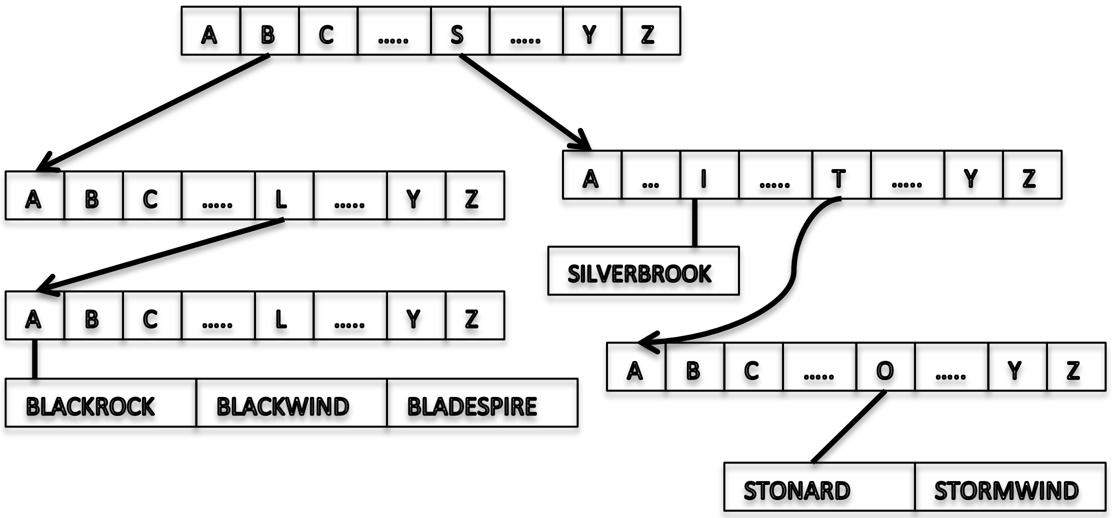
BLADESPIRE



Next is to insert BLADESPIRE and we can easily traverse through the path B → L → A. Since the maximum depth has been reached, we skimpily insert BLADESPIRE into _words at the 3rd level.

Note: *you need to keep _words sorted when you insert the new string. You will keep it sorted by insert the keywords into proper positions. You cannot use any sorting methods to keep it sorted.*

STONARD



The last example here is to insert STONARD. First we go to 'S' (`_tnode[18]`) at the first level and check STONARD with the strings stored in _words. You will find that STONARD and STORMWIND have 3 prefix characters in common, which means we need to create new levels. There's only one string with the prefix "SI", so we do not need to create a new level for SILVERBROOK. STONARD and STORMWIND go to 'O' at the 3rd level as previous example.

Implementing ttree Class

There are two classes need to be implemented for `ttree` class, `tnode` and `ttree` classes. `tnode` class is used to store a node in the tree while `ttree` is the tree itself. These two classes are as follows.

```
class tnode
{
private:
   ttree* _nextLevel;  // pointer to the ttree at the next level
   list<string>* _words;  // store keywords
public:
   tnode();
   ~tnode();
   // ostream operator and other methods
};

class ttree
{
private:
   tnode* _tnodes;  // pointer to _tnodes, a tnode
                 // array of size 26 will be created
                 // in the constructor
   int _maxDepth;  // max depth allowance
   int _currentDepth; // the depth of current node ( root is depth 1)
public:
   ttree();  // set the maxDepth to 5 in the empty constuctor
   ttree( int maxDepth);  // constructor with given maxDepth value
   ~ttree();
   void insert(string key);  // insert 'key' into the tree
   void search(string key);  // search for 'key' in the tree and
                          // print out all the matches
   // ostream operator and other methods
};
```

In this project, only 3 methods (other than the constructors/destructors) are required. However, you will need to create other methods that help you to finish the tasks involved in these methods. You are not allowed to add additional data members to these classes. The detail of the 3 required methods are as follows.

insert

`insert()` method works as described in the example. You may want to split the tasks need to finish the insertion in different cases such as “create a new level”, “determine if a new level need to be created”, etc. Note that when you create a new level, you will need to pass _maxDepth and _currentDepth to the next level so the new level knows its depth and the limitation.

search

`search()` method will "print out" all the keywords stored in the tree that their prefixes match the given key. For example, BLACKROCK BLACKWIND BLADSPIRE will be printed out if the key is "BLA". If the key is BLAC, BLACKROCK, BLACKWIND will be printed out. If the key is "S", everything in the 'S' subtree will printed out: SILVERWIND STONARD STORMWIND. If there's no matches, throw `ttreeNotFound()`. Your `main()` should catch this exception.

ostream operator

In ostream operator, the whole `ttree` will be output to ostream. The format rules are as follows:

1. Preorder traversal is used.
2. There will be `3*(_currentLevel -1)` spaces when the content of a level is printing out.
3. You will need to print the corresponding letter (0→ A, 1→ B ……) of the tnode before printing out the content. If a tnode's `_nextLevel` and `_words` are both node, nothing will be printed (including the corresponding letter of the tnode)
4. Each tnode uses new line for displaying, but for the `_words` array are with its corresponding tnode.

The ostream output of the example is shown in the following box. Space is replaced by underscore (_) so you can identify them easily. The texts in green color are comments

```
# A is not shown because both pointers are NULL
B                                     # level 1
___L                                  # level 2, 3 leading spaces
______A BLACKROCK BLACKWIND BLADESPIRE # A and _words
S
___I SILVERWIND
___T
______O STONARD STORMWIND
```

Cross-Reference

Note there are cross-references in these two classes. So you need to tell the compiler the existence of these two classes before you declare them in other classes.

```
#include <iostream>
......

class tnode;
class ttree;

class tnode
{
private:
   ttree* _nextLevel;
......
};

class ttree
{
private:
   ArrayClass<tnode>* _tnodes
......
};
```

main() and the input file

You will write the `main()` to read the input file. Each line of the input represents a command. The commands that will appear in the input files are:

1. `I` *`keyword`*: insert keyword into the tree.
2. `S` *`keyword`*: search and print out the strings stored in the tree that match keyword.
3. D: use ostream operator to print out the tree.

There will be no limitation on the size of the input file, process it until EOF is read. A sample input is provided below and a larger one will be posted later this week.

```
I ALDRASSIL
I STILLWHISPER
I AUCHINDOUN
I DALARAN
I STONEWATCH
D
S ST
I BLACKWOOD
I BLADESPIRE
I STILLPINE
I DARROWSHIRE
D
```

```
S ST
S BLAC
I DREADMAUL
I STILLWATER
I STONARD
D
S ST
S A
I AMBERMILL
I GADGETZAN
I BLACKROCK
I ANVILMAR
I SYLVANAAR
I BLACKWIND
I DARKSHIRE
I BLADEFIST
D
S AN
S BLAD
```

Constraints

1. In this project, the header files you can use are <iostream>, <string> and <list>. No other libraries will be used/allowed. You can use the classes you created from previous projects.
2. You are NOT allowed to add any data members to the classes. However, you can create additional private methods to simplify your work in implementing the methods.
3. None of the projects will be a group project. Consulting with other members of this class on programming projects is strictly not allowed and plagiarism charges will be imposed on students who do not follow this.